

Negation in SPARQL

Renzo Angles^{1,3} and Claudio Gutierrez^{2,3}

¹ Dept. of Computer Science, Universidad de Talca, Chile

² Dept. of Computer Science, Universidad de Chile, Chile

³ Center for Semantic Web Research

Abstract. This paper presents a thorough study of negation in SPARQL. The types of negation supported in SPARQL are identified and their main features discussed. Then, we study the expressive power of the corresponding negation operators. At this point, we identify a core SPARQL algebra which could be used instead of the W3C SPARQL algebra. Finally, we analyze the negation operators in terms of their compliance with elementary axioms of set theory.

1 Introduction

The notion of negation has been largely studied in database query languages, mainly due to their implications in aspects of expressive power and computational complexity [7–9, 15, 19]. There have been proposed several and distinct types of negation, and it seems difficult to get agreement about a standard one. Such heterogeneity comes from intrinsic properties and semantics of each language, features that determine its ability to support specific type(s) of negation(s). For instance, rule-based query languages (e.g. Datalog) usually implement “negation by failure”, algebraic query languages (e.g. the relational algebra) usually include a Boolean-like difference operator, and SQL-like query languages support several versions of negation, and operators that combine negation with subqueries (e.g. NOT EXISTS).

The case of SPARQL is no exception. SPARQL 1.0 did not have any explicit form of negation in its syntax, but could simulate negation by failure. For the next version, SPARQL 1.1, the incorporation of negation generated a lot of debate. As result, SPARQL 1.1 provides four types of negation: negation of filter constraints, by using the Boolean NOT operator; negation as failure, implemented as the combination of an optional graph pattern and the bound operator; difference of graph patterns, expressed by the MINUS operator; and existential negation of graph patterns, expressed by the NOT-EXISTS operator.

The main positive and negative aspects of these types of negation are remarked in the SPARQL specifications, and more detailed discussions are available in the records of the SPARQL working group⁴. However, to the best of our knowledge, there exists no formal study about negation in SPARQL. This is the goal we tackle throughout this document.

⁴ <https://www.w3.org/2009/sparql/wiki/Design:Negation>

In this paper we present a thorough study of negation in SPARQL. First, we formalize the syntax and semantics of the different types of negation supported in SPARQL, and discuss their main features. Then, we study the relationships (in terms of expressive power) among the negation operators, first at the level of the SPARQL algebra, and subsequently at the level of SPARQL graph patterns. Finally, we present a case-by-case analysis of the negation operations with respect to their compliance with elementary axioms of set theory.

It would be good to have a simple version of negation operators that conform to a known intuition. This is the other main contribution of this paper. First we introduce the DIFF operator as another way of expressing the negation of graph patterns (it is the SPARQL version of the EXCEPT operator of SQL). The semantics of DIFF is based on a simple difference operator introduced at the level of the SPARQL algebra. With this new difference operator, we show that one can define a core algebra (i.e. projection, selection, join, union and simple difference) which is able to express the W3C SPARQL algebra. We show that this algebra is also able to define the SPARQL operators, thus it can be considered a sort of “core” SPARQL algebra. Additionally, we show that the DIFF operator behaves well regarding its compliance with well known axioms of set theory.

Organization of the paper. The syntax and semantics of SPARQL graph patterns are presented in Section 2. In Section 3, we discuss the main characteristics of the types of negation supported by SPARQL. The expressive power of the negation operators is studied in Section 4. In Section 5, we analyze set-theoretic properties of two negation operators. Finally, some conclusions are presented in Section 6.

2 SPARQL graph patterns

The following definition of SPARQL graph patterns is based on the formalism used in [16], but in agreement with the W3C SPARQL specifications [17, 10]. Particularly, graph patterns will be studied assuming bag semantics (i.e. allowing duplicates in solutions).

RDF graphs. Assume two disjoint infinite sets I and L , called IRIs and literals respectively. An *RDF term* is an element in the set $T = I \cup L$ ⁵. An *RDF triple* is a tuple $(v_1, v_2, v_3) \in I \times I \times T$ where v_1 is the *subject*, v_2 the *predicate* and v_3 the *object*. An *RDF Graph* (just graph from now on) is a set of RDF triples. The *union* of graphs, $G_1 \cup G_2$, is the set theoretical union of their sets of triples. Additionally, assume the existence of an infinite set V of variables disjoint from T . We will use $\text{var}(\alpha)$ to denote the set of variables occurring in the structure α .

⁵ In addition to I and L , RDF and SPARQL consider a domain of anonymous resources called blank nodes. The occurrence of blank nodes introduces several issues that are not discussed in this paper. Based on the results presented in [11], we avoid the use of blank nodes assuming that their absence does not largely affect the results presented in this paper.

p	q	$p \wedge q$	$p \vee q$		
<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>		
<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>		
<i>true</i>	<i>error</i>	<i>error</i>	<i>true</i>		
<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>	p	$\neg p$
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>error</i>	<i>false</i>	<i>error</i>	<i>false</i>	<i>true</i>
<i>error</i>	<i>true</i>	<i>error</i>	<i>true</i>	<i>error</i>	<i>error</i>
<i>error</i>	<i>false</i>	<i>false</i>	<i>error</i>		
<i>error</i>	<i>error</i>	<i>error</i>	<i>error</i>		

Table 1. Three-valued logic for evaluating selection formulas.

A *solution mapping* (or just *mapping* from now on) is a partial function $\mu : V \rightarrow T$ where the domain of μ , $\text{dom}(\mu)$, is the subset of V where μ is defined. The *empty mapping*, denoted μ_0 , is the mapping satisfying that $\text{dom}(\mu_0) = \emptyset$. Given $?X \in V$ and $c \in T$, we use $\mu(?X) = c$ to denote the solution mapping variable $?X$ to term c . Similarly, $\mu_{?X \rightarrow c}$ denotes a mapping μ satisfying that $\text{dom}(\mu) = \{?X\}$ and $\mu(?X) = c$. Given a finite set of variables $W \subset V$, the restriction of a mapping μ to W , denoted $\mu|_W$, is a mapping μ' satisfying that $\text{dom}(\mu') = \text{dom}(\mu) \cap W$ and $\mu'(?X) = \mu(?X)$ for every $?X \in \text{dom}(\mu) \cap W$. Two mappings μ_1, μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, when for all $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ it satisfies that $\mu_1(?X) = \mu_2(?X)$, i.e., when $\mu_1 \cup \mu_2$ is also a mapping. Note that two mappings with disjoint domains are always compatible, and that the empty mapping μ_0 is compatible with any other mapping.

A *selection formula* is defined recursively as follows: (i) If $?X, ?Y \in V$ and $c \in I \cup L$ then $(?X = c)$, $(?X = ?Y)$ and $\text{bound}(?X)$ are atomic selection formulas; (ii) If F and F' are selection formulas then $(F \wedge F')$, $(F \vee F')$ and $\neg(F)$ are boolean selection formulas. The evaluation of a selection formula F under a mapping μ , denoted $\mu(F)$, is defined in a three-valued logic (i.e. with values *true*, *false*, and *error*) as follows:

- If F is $?X = c$ and $?X \in \text{dom}(\mu)$, then $\mu(F) = \text{true}$ when $\mu(?X) = c$ and $\mu(F) = \text{false}$ otherwise. If $?X \notin \text{dom}(\mu)$ then $\mu(F) = \text{error}$.
- If F is $?X = ?Y$ and $?X, ?Y \in \text{dom}(\mu)$, then $\mu(F) = \text{true}$ when $\mu(?X) = \mu(?Y)$ and $\mu(F) = \text{false}$ otherwise. If either $?X \notin \text{dom}(\mu)$ or $?Y \notin \text{dom}(\mu)$ then $\mu(F) = \text{error}$.
- If F is $\text{bound}(?X)$ and $?X \in \text{dom}(\mu)$ then $\mu(F) = \text{true}$ else $\mu(F) = \text{false}$.
- If F is a complex selection formula then it is evaluated following the three-valued logic presented in Table 1.

A *multiset* (or *bag*) of solution mappings is an unordered collection in which each solution mapping may appear more than once. A multiset will be represented as a set of solution mappings, each one annotated with a positive integer which defines its acity (i.e. its cardinality). We use the symbol Ω to denote a multiset and $\text{card}(\mu, \Omega)$ to denote the cardinality of the mapping μ in the multiset

Ω . In this sense, it applies that $\text{card}(\mu, \Omega) = 0$ when $\mu \notin \Omega$. We use Ω_0 to denote the multiset $\{\mu_0\}$ such that $\text{card}(\mu_0, \Omega_0) > 0$ (Ω_0 is called the join identity). The domain of a solution mapping Ω is defined as $\text{dom}(\Omega) = \bigcup_{\mu \in \Omega} \text{dom}(\mu)$.

W3C SPARQL algebra. Let Ω_1, Ω_2 be multisets of mappings, W be a set of variables and F be a selection formula. The *SPARQL algebra for multisets of mappings* is composed of the operations of projection, selection, join, difference, left-join, union and minus, defined respectively as follows:

- $\pi_W(\Omega_1) = \{\mu' \mid \mu \in \Omega_1, \mu' = \mu|_W\}$
where $\text{card}(\mu', \pi_W(\Omega_1)) = \sum_{\mu'=\mu|_W} \text{card}(\mu, \Omega_1)$
- $\sigma_F(\Omega_1) = \{\mu \in \Omega_1 \mid \mu(F) = \text{true}\}$
where $\text{card}(\mu, \sigma_F(\Omega_1)) = \text{card}(\mu, \Omega_1)$
- $\Omega_1 \bowtie \Omega_2 = \{\mu = (\mu_1 \cup \mu_2) \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\}$
where $\text{card}(\mu, \Omega_1 \bowtie \Omega_2) = \sum_{\mu=(\mu_1 \cup \mu_2)} \text{card}(\mu_1, \Omega_1) \times \text{card}(\mu_2, \Omega_2)$
- $\Omega_1 \setminus_F \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, (\mu_1 \approx \mu_2) \vee (\mu_1 \sim \mu_2 \wedge (\mu_1 \cup \mu_2)(F) = \text{false})\}$
where $\text{card}(\mu_1, \Omega_1 \setminus_F \Omega_2) = \text{card}(\mu_1, \Omega_1)$
- $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2\}$
where $\text{card}(\mu, \Omega_1 \cup \Omega_2) = \text{card}(\mu, \Omega_1) + \text{card}(\mu, \Omega_2)$
- $\Omega_1 - \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, \mu_1 \approx \mu_2 \vee \text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset\}$
where $\text{card}(\mu_1, \Omega_1 - \Omega_2) = \text{card}(\mu_1, \Omega_1)$
- $\Omega_1 \bowtie_F \Omega_2 = \sigma_F(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus_F \Omega_2)$
where $\text{card}(\mu, \Omega_1 \bowtie_F \Omega_2) = \text{card}(\mu, \sigma_F(\Omega_1 \bowtie \Omega_2)) + \text{card}(\mu, \Omega_1 \setminus_F \Omega_2)$

Syntax of SPARQL graph patterns. A SPARQL *graph pattern* is defined recursively as follows:

- A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a graph pattern called a *triple pattern*.⁶
- If P_1 and P_2 are graph patterns then $(P_1 \text{ AND } P_2)$, $(P_1 \text{ UNION } P_2)$, $(P_1 \text{ OPT } P_2)$, $(P_1 \text{ MINUS } P_2)$ and $(P_1 \text{ NOT-EXISTS } P_2)$ are graph patterns.
- If P_1 is a graph pattern and C is a filter constraint (as defined below) then $(P_1 \text{ FILTER } C)$ is a graph pattern.

A *filter constraint* is defined recursively as follows: (i) If $?X, ?Y \in V$ and $c \in I \cup L$ then $(?X = c)$, $(?X = ?Y)$ and $\text{bound}(?X)$ are *atomic filter constraints*; (ii) If C_1 and C_2 are filter constraints then $(!C_1)$, $(C_1 \parallel C_2)$ and $(C_1 \&\& C_2)$ are *complex filter constraints*. Given a filter constraint C , we denote by $f(C)$ the selection formula obtained from C . Note that there exists a simple and direct translation from filter constraints to selection formulas and viceversa.

Given a triple pattern t and a mapping μ such that $\text{var}(t) \subseteq \text{dom}(\mu)$, we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ . Overloading the above definition, we denote by $\mu(P)$ the graph pattern obtained by the recursive substitution of variables in every triple pattern and filter constraint occurring in the graph pattern P according to μ .

⁶ We assume that any triple pattern contains at least one variable.

Semantics of SPARQL graph patterns. The evaluation of a SPARQL graph pattern P over an RDF graph G is defined as a function $\llbracket P \rrbracket_G$ (or $\llbracket P \rrbracket$ where G is clear from the context) which returns a multiset of solution mappings. Let P_1, P_2, P_3 be graph patterns and C be a filter constraint. The evaluation of a graph pattern P over a graph G is defined recursively as follows:

1. If P is a triple pattern t , then $\llbracket P \rrbracket_G = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \wedge \mu(t) \in G\}$ where each mapping μ has cardinality 1.
2. $\llbracket (P_1 \text{ AND } P_2) \rrbracket_G = \llbracket P_1 \rrbracket_G \bowtie \llbracket P_2 \rrbracket_G$
3. If P is $(P_1 \text{ OPT } P_2)$ then
 - (a) if P_2 is $(P_3 \text{ FILTER } C)$ then $\llbracket P \rrbracket_G = \llbracket P_1 \rrbracket_G \bowtie_{f(C)} \llbracket P_3 \rrbracket_G$
 - (b) else $\llbracket P \rrbracket_G = \llbracket P_1 \rrbracket_G \bowtie_{(true)} \llbracket P_2 \rrbracket_G$
4. $\llbracket (P_1 \text{ MINUS } P_2) \rrbracket_G = \llbracket P_1 \rrbracket_G - \llbracket P_2 \rrbracket_G$
5. $\llbracket (P_1 \text{ NOT-EXISTS } P_2) \rrbracket_G = \{\mu \mid \mu \in \llbracket P_1 \rrbracket_G \wedge \llbracket \mu(P_2) \rrbracket_G = \emptyset\}$
6. $\llbracket (P_1 \text{ UNION } P_2) \rrbracket_G = \llbracket P_1 \rrbracket_G \cup \llbracket P_2 \rrbracket_G$
7. $\llbracket (P_1 \text{ FILTER } C) \rrbracket_G = \sigma_{f(C)}(\llbracket P_1 \rrbracket_G)$

3 Types of negation in SPARQL

We can distinguish four types of negation in SPARQL: negation of filter constraints, negation as failure, negation by MINUS and negation by NOT-EXISTS. The main features of these types of negation will be discussed in this section.

Negation of filter constraints. The most basic type of negation in SPARQL is the one allowed in filter graph patterns by including constraints of the form $(!C)$. Following the semantics of SPARQL, a graph pattern $(P \text{ FILTER } (!C))$, returns the mappings μ in $\llbracket P \rrbracket$ such that μ satisfies the filter constraint $(!C)$, i.e. μ does not satisfy the constraint C .

Considering that the evaluation of a graph pattern could return mappings with unbound variables (similar to NULL values in SQL), SPARQL uses a three-valued logic for evaluating filter graph patterns, i.e. the evaluation of a filter constraint C can result in *true*, *false* or *error*. For instance, given a mapping μ , the constraint $?X = 1$ evaluates to *true* when $\mu(?X) = 1$, *false* when $\mu(?X) \neq 1$, and *error* when $?X \notin \text{dom}(\mu)$. Note that FILTER eliminates any solutions that result in *false* or *error*. Recalling the semantics defined in Table 1, we have that $(!C)$ evaluates to *true* when C is *false*, *false* when C is *true*, and *error* when C is *error*. This type of negation, called *strong negation* [20], allows to deal with incomplete information in a similar way to the NOT operator of SQL.

The negation of filter constraints is a feature well established in SPARQL and does not deserve major discussion. In the rest of the paper we concentrate our interest on the negation of graph patterns.

Negation as failure. SPARQL 1.0 does not include an operator to express the negation of graph patterns. In an intent of patching this issue, the SPARQL specification remarks that the negation of graph patterns can be implemented

as a combination of an optional graph pattern and a filter constraint containing the bound operator (see [17], Sec. 11.4.1). This style of negation, called *negation as failure* in logic programming, can be illustrated as a graph pattern P of the form $((P_1 \text{ OPT } P_2) \text{ FILTER } (! \text{ bound } (?X)))$ where $?X$ is a variable of P_2 not occurring in P_1 . Note that, the evaluation of P returns the mappings of $\llbracket (P_1 \text{ OPT } P_2) \rrbracket$ satisfying that variable $?X$ is unbounded, i.e. $?X$ does not match P_2 . In other words, P returns “the solution mappings of P_1 that are not compatible with the solutions mappings of P_2 ”. Unfortunately, this simulation does not work for the general case [5]. A discussion of the issues and the general solution for implementing negation by failure is included in Appendix A.

In order to facilitate the study of negation by failure in SPARQL, we introduce the operator DIFF as an explicit way of expressing it.

Definition 1 (DIFF). *Let P_1 and P_2 be graph patterns. The DIFF operator is defined as $\llbracket (P_1 \text{ DIFF } P_2) \rrbracket = \{\mu_1 \in \llbracket P_1 \rrbracket \mid \forall \mu_2 \in \llbracket P_2 \rrbracket, \mu_1 \not\sim \mu_2\}$.*

It is very important to note that the DIFF operator is not defined in SPARQL 1.0 nor in SPARQL 1.1. However, it can be directly implemented with the difference operator of the algebra of solution mappings.

Negation by MINUS. SPARQL 1.1 introduced the MINUS operator as an explicit way of expressing the negation (or difference) of graph patterns. Note that DIFF and MINUS have similar definitions. The difference is given by the restriction about disjoint mappings included by the MINUS operator. Such restriction, named Antijoin Restriction inside the SPARQL working group⁷, was introduced to avoid solutions with vacuously compatible mappings. Such restriction causes different results for DIFF and MINUS. Basically, if P_1 and P_2 do not have variables in common then $\llbracket (P_1 \text{ DIFF } P_2) \rrbracket = \emptyset$ whereas $\llbracket (P_1 \text{ MINUS } P_2) \rrbracket = \llbracket P_1 \rrbracket$.

Note that both, DIFF and MINUS resemble the EXCEPT operator of SQL[14], i.e. given two SQL queries Q_1 and Q_2 , the expression $(Q_1 \text{ EXCEPT } Q_2)$ allows to return all rows that are in the table obtained from Q_1 *except* those that also appear in the table obtained from Q_2 . Considering that EXCEPT makes reference to the difference of two relations (tables), we can say that DIFF and MINUS allow to express the *difference of two graph patterns*.

Negation by NOT-EXISTS. Another type of negation defined in SPARQL 1.1 is given by the NOT-EXISTS operator. The main feature of this type of negation is the possible occurrence of correlation. Given a graph pattern $P = (P_1 \text{ NOT-EXISTS } P_2)$, we will say that P_1 and P_2 are *correlated* when $\text{var}(P_1) \cap \text{var}(P_2) \neq \emptyset$, i.e. there exist variables occurring in both P_1 and P_2 , and such variables are called *correlated variables*. In this case, the evaluation of P is attained by replacing variables in P_2 with the corresponding values given by the current mapping μ of $\llbracket P_1 \rrbracket$, and testing whether the evaluation of the graph

⁷ <http://lists.w3.org/Archives/Public/public-rdf-dawg/2009JulSep/0030.html>

pattern $\mu(P_2)$ returns no solutions. This way of evaluating correlated queries is based on the nested iteration method [13] of SQL. The correlation of variables in NOT-EXISTS introduces several issues that have been studied in the context of subqueries in SPARQL [3, 4]. Some of these issues are discussed in Section 4.2.

DIFF, MINUS and NOT-EXISTS are three different ways of expressing negation in SPARQL. DIFF and MINUS represent the difference of two graph patterns whereas NOT-EXISTS tests the presence of a graph pattern.

4 Expressive Power

Let us recall some definitions related to expressive power. By the expressive power of a query language, we understand the set of all queries expressible in that language [1]. We will say that an operator O is expressible in a language L iff a subset of the operators of L allow to express the same queries as O . Finally, a language L contains a language L' iff every operator of L' is expressible by L .

In this section we study the expressive power of the negation operators defined in the above section, i.e. DIFF, MINUS and NOT-EXISTS. First, we introduce a “core” SPARQL algebra which contains the W3C SPARQL algebra, but having the advantage of being smaller and simpler. Such core algebra is the basis to define equivalences among graph pattern operators, in particular those implementing negation.

4.1 The core SPARQL algebra

Let us introduce a “core” algebra for SPARQL which is able to express all the high-level operators of the W3C SPARQL language. Recall that the W3C SPARQL algebra, defined in Section 2, is composed by the operators of *projection*, *selection*, *join*, *union*, *left-join* and *minus*. Our core algebra is based on a new operator called “simple difference”.

Definition 2 (Simple difference). *The simple difference between two solution mappings, Ω_1 and Ω_2 , is defined as $\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, \mu_1 \not\sim \mu_2\}$ where $\text{card}(\mu_1, \Omega_1 \setminus \Omega_2) = \text{card}(\mu_1, \Omega_1)$.*

Definition 3 (Core SPARQL algebra). *The core SPARQL algebra is composed by the operations of projection, selection, join, union and simple difference.*

Next, we will show that the core SPARQL algebra contains the W3C SPARQL algebra. Specifically, we will show that the operators of *difference*, *left-join* and *minus* (of the W3C SPARQL algebra) can be simulated with the *simple difference* operator (of the core SPARQL algebra).

Lemma 1. *The difference operator (of the W3C SPARQL algebra) is expressible in the core SPARQL algebra.*

Proof. Recall that the *difference* operator is defined as $\Omega_1 \setminus_F \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, (\mu_1 \approx \mu_2) \vee (\mu_1 \sim \mu_2 \wedge (\mu_1 \cup \mu_2)(F) = \text{false})\}$. Assume that $\Omega'_1 = \Omega_1 \setminus (\Omega_1 \setminus_F \Omega_2)$, i.e. Ω'_1 is the complement of $\Omega_1 \setminus_F \Omega_2$. We have that $\mu_1 \in \Omega'_1$ iff there exists a mapping $\mu_2 \in \Omega_2$ satisfying that $\mu_1 \sim \mu_2$ and $(\mu_1 \cup \mu_2)(F)$ evaluates to either *true* or *error*. Hence, we will have that

$$\Omega_1 \setminus_F \Omega_2 = \Omega_1 \setminus (\sigma_F(\Omega_1 \bowtie \Omega_2) \cup ((\Omega_1 \bowtie \Omega_2) \setminus (\sigma_{(F \vee \neg F)}(\Omega_1 \bowtie \Omega_2))))$$

where $\sigma_F(\Omega_1 \bowtie \Omega_2)$ corresponds to the mappings evaluating F as *true*, and $((\Omega_1 \bowtie \Omega_2) \setminus (\sigma_{(F \vee \neg F)}(\Omega_1 \bowtie \Omega_2)))$ corresponds to the mappings evaluating F as *error*.

Note that the above equivalence is based on a faithful interpretation as occurs in the current specification. However, the document of errata in SPARQL 1.1 includes a clarification that differs of our interpretation⁸: “The definition of Diff should more clearly say that expressions that evaluate with an error are considered to be false.” Under this interpretation, the proof is much simpler⁹:

$$\Omega_1 \setminus_F \Omega_2 = \Omega_1 \setminus (\sigma_F(\Omega_1 \bowtie \Omega_2)).$$

Lemma 2. *The left-join operator (of the W3C SPARQL algebra) is expressible in the core SPARQL algebra.*

Proof. Recall that $\Omega_1 \bowtie_F \Omega_2 = \sigma_F(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus_F \Omega_2)$. Considering that the *difference* expression $(\Omega_1 \setminus_F \Omega_2)$ can be expressed with *simple difference* (Lemma 1), we have that

$$\Omega_1 \bowtie_F \Omega_2 = \sigma_F(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus (\sigma_F(\Omega_1 \bowtie \Omega_2)))$$

Lemma 3. *The minus operator (of the W3C SPARQL algebra) is expressible in the core SPARQL algebra.*

Proof. Note that the difference between $\Omega_1 - \Omega_2$ and $\Omega_1 \setminus \Omega_2$ is given by the condition $\text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset$. Intuitively, it applies that $(\Omega_1 - \Omega_2) = \Omega_1$ when $\text{dom}(\Omega_1) \cap \text{dom}(\Omega_2) = \emptyset$. In the case that $\text{dom}(\Omega_1) \cap \text{dom}(\Omega_2) \neq \emptyset$, the rewriting is a little more tricky.

Recall that the operation $\Omega_1 - \Omega_2$ is defined by the set $S = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, \mu_1 \approx \mu_2 \vee \text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset\}$.

⁸ <https://www.w3.org/2013/sparql-errata#errata-query-12>

⁹ In an earlier version of this report, the proof of Lemma 1 was based on the full definition of LeftJoin presented in Sec. 18.5 of the current SPARQL 1.1 specification, a statement that was reported to be wrong in the errata <http://www.w3.org/2013/sparql-errata#errata-query-7a> (We thank to R. Kontchakov, who called to our attention this issue in a personal communication, May 19th, 2016.) The same bug in the proof of Lemma 1 was included in the article “Negation in SPARQL” (Alberto Mendelzon International Workshop on Foundations of Data Management, AMW’2016). Fortunately, the problem was with the proof, not with the statement of the Lemma which remains untouched and thus does not affect the rest of results in such article nor in this report.

Note that the complement of S in Ω_1 is the set

$$S^c = \{\mu_1 \in \Omega_1 \mid \exists \mu_2 \in \Omega_2, \mu_1 \sim \mu_2 \wedge \text{dom}(\mu_1) \cap \text{dom}(\mu_2) \neq \emptyset\}.$$

Assuming that $\text{dom}(\Omega_1) \cap \text{dom}(\Omega_2) = \{?X_1, \dots, ?X_n\}$, $(\Omega_1 - \Omega_2)$ can be simulated with an expression of the form $\Omega_1 \setminus (\sigma_F(\Omega_1 \bowtie \Omega_3))$ where Ω_3 is a copy of Ω_2 where every variable X_i has been replaced with a free variable X'_i , and F is a selection formula of the form $(\dots (?X_1 = ?X'_1 \wedge ?X_2 = ?X'_2) \dots) \wedge ?X_n = ?X'_n$. Note that, the expression $\sigma_F(\Omega_1 \bowtie \Omega_3)$ returns the set S^c , which is subtracted from Ω_1 to obtain S . It is easy to see that the proposed equivalent expression preserves the cardinalities. Hence, we have proved that $(\Omega_1 - \Omega_2)$ can be simulated with $(\Omega_1 \setminus \Omega_2)$.

Based on Lemmas 1, 2 and 3, we can present our main result about the expressive power of the core SPARQL algebra.

Theorem 1. *The core SPARQL algebra is equivalent with the W3C SPARQL algebra.*

This result implies that the W3C SPARQL algebra could be implemented by a subset of the original operators (projection, selection, join and union), plus the simple difference operator.

4.2 SPARQL^{DIFF}: A core fragment with negation

Let us redefine the DIFF operator by using the simple difference operator of the core algebra as follows:

$$\llbracket (P_1 \text{ DIFF } P_2) \rrbracket = \llbracket P_1 \rrbracket \setminus \llbracket P_2 \rrbracket.$$

Assume that SPARQL^{DIFF} is the language defined (recursively) by graph patterns of the form $(P \text{ AND } P')$, $(P \text{ UNION } P')$, $(P \text{ DIFF } P')$ and $(P \text{ FILTER } C)$. Next, we will show that SPARQL^{DIFF} can express a fragment of SPARQL whose NOT-EXISTS graph patterns satisfy a restricted notion of correlated variables.

Expressing OPTIONAL and MINUS with DIFF Here, we show that the OPT and MINUS operators can be expressed in SPARQL^{DIFF} by using the DIFF operator.

Lemma 4. *The OPT operator is expressible in SPARQL^{DIFF}.*

Proof. Recall the semantics of optional graph patterns as presented in Section 2. Given a graph pattern P of the form $(P_1 \text{ OPT } P_2)$, we have two cases:

- (i) if P_2 is $(P_3 \text{ FILTER } C)$ then $\llbracket P \rrbracket = \llbracket P_1 \rrbracket \bowtie_{f(C)} \llbracket P_3 \rrbracket$;
- (ii) else, $\llbracket P \rrbracket = \llbracket P_1 \rrbracket \bowtie_{(true)} \llbracket P_2 \rrbracket$.

Note that this definition follows an operational semantics in the sense that the evaluation of $(P_1 \text{ OPT } P_2)$ depends on the structure of P_2 , i.e. when P_2 is a filter graph pattern applies case (i), otherwise case (ii). Let us analyze both cases.

Case (i): We have that $\llbracket P_1 \rrbracket \bowtie_{f(C)} \llbracket P_3 \rrbracket = \sigma_F(\Omega_1 \bowtie \Omega_3) \cup (\Omega_1 \setminus_F \Omega_3)$ where $\Omega_1 = \llbracket P_1 \rrbracket$, $\Omega_3 = \llbracket P_3 \rrbracket$ and $F = f(C)$. If we rewrite the right side in terms of simple difference (Lemma 2), we obtain $\sigma_F(\Omega_1 \bowtie \Omega_3) \cup (\Omega_1 \setminus (\sigma_F(\Omega_1 \bowtie \Omega_3)))$. Similarly, the graph pattern $(P_1 \text{ OPT } (P_3 \text{ FILTER } C))$ can be rewritten as

$$(((P_1 \text{ AND } P_3) \text{ FILTER } C) \text{ UNION } (P_1 \text{ DIFF } ((P_1 \text{ AND } P_3) \text{ FILTER } C))).$$

Case (ii): We have that $\llbracket P_1 \rrbracket \bowtie_{(true)} \llbracket P_2 \rrbracket = \sigma_{(true)}(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus_{(true)} \Omega_2)$ where $\Omega_1 = \llbracket P_1 \rrbracket$ and $\Omega_2 = \llbracket P_2 \rrbracket$. If we rewrite the right side in terms of simple difference (Lemma 2), we obtain $\sigma_{(true)}(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus (\sigma_{(true)}(\Omega_1 \bowtie \Omega_2)))$, which can be reduced to $(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus (\Omega_1 \bowtie \Omega_2))$. Similarly, the graph pattern $(P_1 \text{ OPT } P_2)$ can be rewritten as

$$((P_1 \text{ AND } P_2) \text{ UNION } (P_1 \text{ DIFF } (P_1 \text{ AND } P_2))).$$

Hence, we can use the above transformation to simulate the operational semantics of OPT by using DIFF, i.e. OPT is expressible in SPARQL^{DIFF}.

Lemma 5. *The MINUS operator is expressible in SPARQL^{DIFF}.*

Proof. Given a graph pattern $(P_1 \text{ MINUS } P_2)$, we have that:

- (i) If $\text{var}(P_1) \cap \text{var}(P_2) = \emptyset$ then $\llbracket (P_1 \text{ MINUS } P_2) \rrbracket = \llbracket P_1 \rrbracket$.
- (ii) Else, $\llbracket (P_1 \text{ MINUS } P_2) \rrbracket = \Omega_1 - \Omega_2$ where $\Omega_1 = \llbracket P_1 \rrbracket$ and $\Omega_2 = \llbracket P_2 \rrbracket$. Recalling the simulation of $(\Omega_1 - \Omega_2)$ with $(\Omega_1 \setminus \Omega_2)$ presented in Lemma 3, we have that $(P_1 \text{ MINUS } P_2)$ can be rewritten to a graph pattern of the form $(P_1 \text{ DIFF } ((P_1 \text{ AND } P_3) \text{ FILTER } C))$ where P_3 and C must be created by following a procedure similar to the one described in Lemma 3. Hence, we have proved that MINUS can be expressed in SPARQL^{DIFF}.

NOT-EXISTS vs DIFF In order to give a basic idea of the expressive power of NOT-EXISTS, we will analyze its relationship with the DIFF operator. Given the graph patterns $P = (P_1 \text{ NOT-EXISTS } P_2)$ and $P' = (P_1 \text{ DIFF } P_2)$, the basic case of equivalence between P and P' is given when $\text{var}(P_1) \cap \text{var}(P_2) = \emptyset$. Note that, the non occurrence of correlated variables between P_1 and P_2 implies that, both P and P' are restricted to test whether P_2 returns an empty solution.

Intuitively, one can assume that any graph pattern $P = (P_1 \text{ NOT-EXISTS } P_2)$ can be directly translated into $P' = (P_1 \text{ DIFF } P_2)$ such that P and P' are equivalent. This is for example argued by Kaminski et. al [12] (Lemma 3). However, the translation given there does not work. For instance, consider the graph $G = \{(a,p,b), (f,p,b), (c,q,d), (e,r,a)\}$ and the graph patterns $P = ((?X \text{ p } b) \text{ NOT-EXISTS } (?Z \text{ q } d) \text{ NOT-EXISTS } (?W \text{ r } ?X))$ and $P' = ((?X \text{ p } b) \text{ DIFF } (?Z \text{ q } d) \text{ DIFF } (?W \text{ r } ?X))$. In this case, P and P' are not equivalent such that $\llbracket P \rrbracket_G = \{\mu_{?X \rightarrow a}\}$ whereas $\llbracket P' \rrbracket_G = \{\mu_{?X \rightarrow a}, \mu'_{?X \rightarrow f}\}$.

Despite the above negative result, it will be interesting to identify a subset of NOT-EXISTS graph patterns that can be expressed by using the DIFF operator. To do this, we will introduce the notion of “safe” and “unsafe” variables. The set

of *safe variables* in a graph pattern P , denoted $\text{svar}(P)$, is defined recursively as follows: If P is a triple pattern, then $\text{svar}(P) = \text{var}(P)$; If P is $(P_1 \text{ AND } P_2)$ then $\text{svar}(P) = \text{svar}(P_1) \cup \text{svar}(P_2)$; If P is $(P_1 \text{ UNION } P_2)$ or $(P_1 \text{ OPT } P_2)$ then $\text{svar}(P) = \text{svar}(P_1) \cap \text{svar}(P_2)$; If P is $(P_1 \text{ FILTER } C)$, $(P_1 \text{ MINUS } P_2)$, $(P_1 \text{ NOT-EXISTS } P_2)$ or $(P_1 \text{ DIFF } P_2)$ then $\text{svar}(P) = \text{svar}(P_1)$. Therefore, a variable occurring in $\text{svar}(P)$ is called a *safe variable*, otherwise it is considered *unsafe* in P .

Definition 4 ($\text{SPARQL}_{\text{safe}}^{\text{NEX}}$). Define $\text{SPARQL}_{\text{safe}}^{\text{NEX}}$ as the fragment of SPARQL graph patterns satisfying that the occurrence of a subpattern $(P \text{ NOT-EXISTS } P')$ implies that, for every correlated variable $?X$ between P and P' , it holds that $?X \in \text{svar}(P')$.

Note that, $\text{SPARQL}^{\text{NEX}}$ does not allow graph patterns of the form $(P_1 \text{ NOT-EXISTS } (P_2 \text{ NOT-EXISTS } P_3))$ where P_3 contains correlated variables occurring in P_1 but not occurring in P_2 .

Lemma 6. The NOT-EXISTS graph patterns allowed in $\text{SPARQL}_{\text{safe}}^{\text{NEX}}$ are expressible in $\text{SPARQL}^{\text{DIFF}}$.

Proof. Following the definition of $\text{SPARQL}_{\text{safe}}^{\text{NEX}}$, we have that for any graph pattern $(P_1 \text{ NOT-EXISTS } P_2)$ it satisfies that the domain of $\llbracket P_2 \rrbracket$ contains all the correlated variables between P_1 and P_2 . Given such condition, it is easy to see that $\llbracket (P_1 \text{ DIFF } P_2) \rrbracket$ returns the same solutions as $\llbracket (P_1 \text{ NOT-EXISTS } P_2) \rrbracket$.

Based on Lemmas 4, 5 and 6, we can present our main result about the expressive power of the DIFF operator and $\text{SPARQL}^{\text{DIFF}}$.

Theorem 2. $\text{SPARQL}^{\text{DIFF}}$ contains $\text{SPARQL}_{\text{safe}}^{\text{NEX}}$.

5 Properties of the SPARQL negation operators

In this section we evaluate the negation operators in terms of elementary equivalences found in set theory. Specifically, we consider the following axioms concerning set-theoretic differences [18]:

- | | |
|--|--|
| (a) $A \setminus A \equiv \emptyset$ | (g) $A \setminus (A \cap B) \equiv A \setminus B$ |
| (b) $A \setminus \emptyset \equiv A$ | (h) $A \cap (A \setminus B) \equiv A \setminus B$ |
| (c) $\emptyset \setminus A \equiv \emptyset$ | (i) $(A \setminus B) \cup B \equiv A \cup B$ |
| (d) $A \setminus (A \setminus (A \setminus B)) \equiv A \setminus B$ | (j) $(A \cup B) \setminus B \equiv A \setminus B$ |
| (e) $(A \cap B) \setminus B \equiv \emptyset$ | (k) $A \setminus (B \cap C) \equiv (A \setminus B) \cup (A \setminus C)$ |
| (f) $(A \setminus B) \cap B \equiv \emptyset$ | (l) $A \setminus (B \cup C) \equiv (A \setminus B) \cap (A \setminus C)$ |

Our motivation to consider these equivalences is given by the intrinsic nature of expressing negation in SPARQL, i.e. the difference of graph patterns. Moreover, set-theoretic equivalences are supported by other database languages like relational algebra and SQL. Recalling that SPARQL works under bag semantics, we will evaluate the behavior of the negation operators by examining both, set and bag semantics.

Let us define the notion of equivalence between graph patterns. Two graph patterns P_1 and P_2 are equivalent, denoted by $P_1 \equiv P_2$, if it satisfies that $\llbracket P_1 \rrbracket_G = \llbracket P_2 \rrbracket_G$ for every RDF graph G . In order to evaluate the set-based equivalences in the context of SPARQL, we need to assume two conditions:

- (1) As a general rule, a set-based operator requires two “objects” with the same structure (e.g. two tables with the same schema). Such requirement is implicitly satisfied in SPARQL thanks to the definition of solution mappings as partial functions. Basically, if we fix the set of variables V , then we have that for every pair of solution mappings Ω_1 and Ω_2 it satisfies that $\text{dom}(\Omega_1) = \text{dom}(\Omega_2)$, i.e. Ω_1 and Ω_2 have the same domain of variables (even when some variables could be unbounded). Hence, set-based operations like union and difference can be applied to the algebra of solution mappings.
- (2) SPARQL does not provide an explicit operator for intersecting two graph patterns P_1 and P_2 , i.e. the solution mappings which belong both to $\llbracket P_1 \rrbracket$ and $\llbracket P_2 \rrbracket$. Note however, that a graph pattern $(P_1 \text{ AND } P_2)$ resembles the intersection operation (under set-semantics) when $\llbracket P_1 \rrbracket$ and $\llbracket P_2 \rrbracket$ have the same domain of variables. Recalling condition (1), we will assume that the AND operator could be used to play the role of the intersection operator in the set-theoretic equivalences defined above.

Given the above two conditions, we can apply a direct translation from a set-theoretic equivalence to a graph pattern equivalence. Specifically, the set-difference operator will be mapped to a SPARQL negation operator (DIFF, MINUS or NOT-EXISTS), the set-intersection operator will be mapped to AND, and the set-union operator will be replaced by UNION. Considering the specific features of NOT-EXISTS (i.e. correlation of variables), we will restrict our analysis to DIFF and MINUS.

Let P_1 , P_2 , P_3 , P_4 and P_5 be graph patterns satisfying that $\llbracket P_1 \rrbracket = \emptyset$, $\llbracket P_2 \rrbracket = \{\mu_0\}$, $\text{var}(P_3) = \text{var}(P_4)$ and $\text{var}(P_3) \cap \text{var}(P_5) = \emptyset$. By combining the above four graph patterns, we conducted a case-by-case analysis consisting of twenty five cases for equivalences (a)-(j) and one hundred twenty five cases for equivalences (k) and (l). The differences between set and bag semantics were specially considered for equivalences (h), (i), (k) and (l). Next, we present our findings.

DIFF satisfies most equivalences with exception of (h), (i), (k) and (l). Equivalence (h) presents five cases which are not valid under bag semantics, although they are valid under set semantics. A similar condition occurs with ten and seven cases for equivalences (k) and (l) respectively. Additionally, we found ten cases which are not satisfied by equivalence (i).

MINUS does not satisfy equivalences (e) to (l). We found several cases where equivalences (e), (f), (g), (j) and (k) are not satisfied. Similarly, there exist multiple cases where equivalences (h), (i) and (l) do not apply under bag semantics, but works for set semantics. We would like to remark that the “odd results” presented by the MINUS operator arise because the restriction about disjoint solution mappings introduced in its definition.

In summary, we have that each negation operator presents a particular behavior for the axioms studied here. Although none of them was able to satisfy all the axioms, we think that it does not mean that they are badly defined. In fact, the heterogeneity of the operators is a motivation to study their intrinsic properties and to try the definition of a set of desired properties for negation in SPARQL. The details about our case-by-case analysis are included in the Appendix.

6 Conclusions

In this paper we presented a systematic analysis of the types of negation supported in SPARQL 1.0 and SPARQL 1.1. After introducing the standard relational negation (the DIFF operator) we were able to build a core and intuitive algebra (the same as the standard relational algebra) in SPARQL and prove that it is able to define the graph pattern operators.

We think that having a clear understanding of the operators of a language (in this case, the operators of negation of SPARQL) helps both, developers of databases, and users of the query language. We also think that the core language we identified (which is precisely the well known and intuitive relational algebra) is a much easier way to express queries for database practitioners, who learn from the beginning SQL, which now with this new algebra, can be found in the world of SPARQL.

Acknowledgements. R. Angles and C. Gutierrez are founded by the Millennium Nucleus Center for Semantic Web Research under Grant NC120004.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Angles, R., Gutierrez, C.: The Expressive Power of SPARQL. In: Proceedings of the 7th International Semantic Web Conference (ISWC). pp. 114–129. No. 5318 in LNCS (2008)
3. Angles, R., Gutierrez, C.: SQL Nested Queries in SPARQL. In: Proc. of the 4th Alberto Mendelzon Workshop on Foundations of Data Management (2010)
4. Angles, R., Gutierrez, C.: Subqueries in SPARQL. In: Proc. of the 5th Alberto Mendelzon Workshop on Foundations of Data Management (2011)
5. Angles, R., Gutierrez, C.: Negation in SPARQL. Talk at 8th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW) (2014)
6. Arenas, M., Pérez, J.: Querying Semantic Web Data with SPARQL. In: 30th ACM Symposium on Principles of Database Systems (PODS) (2011)
7. Bárány, V., Cate, B., Segoufin, L.: Guarded negation. *Journal of the ACM* 62(3), 22:1–22:26 (2015)
8. Bidoit, N.: Negation in rule-based database languages: A survey. *Theoretical Computer Science* 78(1), 3–83 (1991)
9. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9(3), 365–386 (1991)

10. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language - W3C Recommendation. <http://www.w3.org/TR/2013/REC-sparql11-query-20130321/> (March 21 2013)
11. Hogan, A., Arenas, M., Mallea, A., Polleres, A.: Everything you always wanted to know about blank nodes. *Journal of Web Semantics* 27(1) (2014)
12. Kaminski, M., Kostylev, E.V., Cuenca Grau, B.: Semantics and Expressive Power of Subqueries and Aggregates in SPARQL 1.1. In: *Proc. of the International Conference on World Wide Web*. pp. 227–238 (2016)
13. Kim, W.: On optimizing an SQL-like nested query. *ACM Transactions on Database Systems (TODS)* 7(3), 443–469 (1982)
14. Melton, J., Simon, A.R.: *SQL:1999 Understanding Relational Language Components*. Morgan Kaufmann (May 2001)
15. Naqvi, S.A.: Negation as failure for first-order queries. In: *Proc. of the Symposium on Principles of Database Systems*. pp. 114–122. ACM (1986)
16. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems (TODS)* 34(3), 1–45 (2009)
17. Prud’hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation. <http://www.w3.org/TR/2008/REC-115-sparql-query-20080115/> (January 15 2008)
18. Suppes, P.: *Axiomatic Set Theory*. The University Series in Undergraduate Mathematics, D. Van Nostrand Company, Inc. (1960)
19. Wagner, G.: A database needs two kinds of negation. In: *Proc. of the Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems*. pp. 357–371 (1991)
20. Wagner, G.: Web rules need two kinds of negation. In: *Proc. of Int. Workshop on Principles and Practice of Semantic Web Reasoning*. pp. 33–50. Springer Berlin Heidelberg (2003)

A Simulation of Negation as Failure in SPARQL

In order to conduct our discussion about the simulation of negation by failure in SPARQL, we need to introduce two concepts: RDF Datasets and the GRAPH operator.

An *RDF dataset* D is a set $\{G_0, \langle u_1, G_1 \rangle, \dots, \langle u_n, G_n \rangle\}$ where each G_i is a graph and each u_j is an IRI. G_0 is called the *default graph* of D and it is denoted $\text{dg}(D)$. Each pair $\langle u_i, G_i \rangle$ is called a *named graph*; define $\text{name}(G_i)_D = u_i$ and $\text{gr}(u_i)_D = G_i$. The set of IRIs $\{u_1, \dots, u_n\}$ is denoted $\text{names}(D)$. Every dataset satisfies that: (i) it always contains one default graph (which could be empty); (ii) there may be no named graphs; (iii) each u_j is distinct; and (iv) $\text{blank}(G_i) \cap \text{blank}(G_j) = \emptyset$ for $i \neq j$. Finally, the *active graph* of D is the graph G_j used for querying D .

Next, we also extend the graph pattern evaluation function. The evaluation of a graph pattern P over a dataset D with active graph G will be denoted as $\llbracket P \rrbracket_G^D$ (or $\llbracket P \rrbracket_G$ where D is clear from the context). Specifically, $\llbracket P \rrbracket_G^D$ returns the multiset of mappings that matches the dataset D according to the graph pattern P .

Let us extend the (recursive) definition of graph pattern. Given a graph pattern P and $n \in I \cup V$, we have that $(n \text{ GRAPH } P)$ is also a graph pattern. The semantics of $(n \text{ GRAPH } P)$ is given as follows:

1. If $u \in I$ then $\llbracket (u \text{ GRAPH } P) \rrbracket^D = \llbracket P \rrbracket_G^D$ where $G = \text{gr}(u)_D$
2. If $?X \in V$ then $\llbracket (?X \text{ GRAPH } P) \rrbracket^D = \bigcup_{v \in \text{names}(D)} (\llbracket P \rrbracket_{\text{gr}(v)_D}^D \bowtie \{\mu ?X \rightarrow v\})$

We are ready to begin our analysis. In Section 3, we argued that the Negation by Failure, and also $(P_1 \text{ DIFF } P_2)$, can be simulated by a pattern of the form

$$((P_1 \text{ OPT } P_2) \text{ FILTER } (! \text{ bound } (?X))) \quad (1)$$

where $?X$ is a variable of P_2 not occurring in P_1 . Unfortunately, there are two issues with this solution:

- Variable $?X$ cannot be an arbitrary variable. For example, P_2 could be in turn an optional pattern $(P_3 \text{ OPT } P_4)$ where only variables in P_3 are relevant to evaluate the condition $(! \text{ bound } (?X))$.
- If P_1 and P_2 do not have variables in common, then there is no variable $?X$ to check unboundedness.

Angles and Gutierrez [2] identified these issues and proposed a solution for them. The solution was shown to have a small bug.

Perez's solution. A second approach which fixed the previous bag was presented in [6], where it was proposed that $(P_1 \text{ DIFF } P_2)$ can be simulated by the pattern

$$((P_1 \text{ OPT } (P_2 \text{ AND } (?X_1 ?X_2 ?X_3))) \text{ FILTER } \neg \text{ bound } (?X_1)) \quad (2)$$

where $?X_1, ?X_2, ?X_3$ are fresh variables mentioned neither in P_1 nor in P_2 .

This solution works well when the empty graph pattern is not allowed in the grammar. In the presence of it, the counter example is given when P_1 and P_2 are empty graph patterns and the default graph is the empty graph. In this case, we have that $\llbracket (P_1 \text{ DIFF } P_2) \rrbracket_{G_0}^D = \emptyset$ whereas $\llbracket (2) \rrbracket_{G_0}^D = \{\Omega_0\}$.

Polleres's solution. A third proposal was sketched by Axel Polleres in the mailing list of the SPARQL W3C Working Group, and discussed with the authors of this paper during the year 2009. The proposed solution is defined formally in the following proposition.

Proposition 1. *Let P_1, P_2 be graph patterns. We have that $(P_1 \text{ DIFF } P_2)$ is equivalent to*

$$(((P_1 \text{ OPT } P_2) \text{ AND } (g \text{ GRAPH } (?X :p :o))) \text{ FILTER } \neg \text{ bound } (?X)) \quad (3)$$

where g is a named graph containing the single triple $(:s :p :o)$ and $?X$ is a free variable.

Proof. In order to prove that the above equivalence holds, we have analyzed the corner cases for $(P_1 \text{ DIFF } P_2)$. In Table 2, we compare the evaluation of the graph patterns presented in Proposition 1, considering all the possible solutions for $\llbracket P_1 \rrbracket$ and $\llbracket P_2 \rrbracket$, and including the special case when the default graph is the empty graph. We showed that the evaluation of both graph patterns are equivalent for all the cases. Additionally, Table 2 includes the results for the graph pattern presented in Equation 2, and shows that it fails in the case (5) when the default graph is the empty graph as described before.

				$G_0 \neq \emptyset$		$G_0 = \emptyset$	
	$\llbracket P_1 \rrbracket$	$\llbracket P_2 \rrbracket$	$\llbracket (P_1 \text{ DIFF } P_2) \rrbracket$	$\llbracket P_3 \rrbracket$	$\llbracket P_4 \rrbracket$	$\llbracket P_3 \rrbracket$	$\llbracket P_4 \rrbracket$
1	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
2	\emptyset	Ω_0	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
3	\emptyset	Ω_2	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
4	Ω_0	\emptyset	Ω_0	Ω_0	Ω_0	Ω_0	Ω_0
5	Ω_0	Ω_0	\emptyset	\emptyset	\emptyset	Ω_0	\emptyset
6	Ω_0	Ω_2	\emptyset	\emptyset	\emptyset	–	–
7	Ω_1	\emptyset	Ω_1	Ω_1	Ω_1	–	–
8	Ω_1	Ω_0	\emptyset	\emptyset	\emptyset	–	–
9	Ω_1	Ω_1	\emptyset	\emptyset	\emptyset	–	–
10	Ω_1	Ω_2	$\Omega_1 \setminus \Omega_2$	$\Omega_1 \setminus \Omega_2$	$\Omega_1 \setminus \Omega_2$	–	–
11	Ω_1	Ω_3	\emptyset	\emptyset	\emptyset	–	–

Table 2. Comparison of two implementations of difference of SPARQL graph patterns. Assume that $\Omega_0 = \{\mu_0\}$ (join identity), and $\Omega_1, \Omega_2, \Omega_3$ are sets of mappings distinct of Ω_0 . Additionally, $\text{dom}(\Omega_1) \cap \text{dom}(\Omega_2) \neq \emptyset$ and $\text{dom}(\Omega_1) \cap \text{dom}(\Omega_3) = \emptyset$. P_3 and P_4 are the graph patterns presented in Equations 2 and 3 respectively. Note that, $\llbracket P_4 \rrbracket$ is equivalent to $\llbracket (P_1 \text{ DIFF } P_2) \rrbracket$ in all the cases, whereas $\llbracket P_3 \rrbracket$ fails when $G_0 = \emptyset$ (i.e. the default graph G_0 is the empty graph).

B Case-by-case analysis

Let P_\emptyset, P_1, P_2 and P_3 be graph patterns satisfying that $\llbracket P_\emptyset \rrbracket = \emptyset$, $\llbracket P_1 \rrbracket \neq \emptyset$, $\llbracket P_2 \rrbracket \neq \emptyset$ and $\llbracket P_3 \rrbracket \neq \emptyset$. Also assume that Ω_1, Ω_2 and Ω_3 are solutions mappings satisfying that $\text{dom}(\Omega_1) \cap \text{dom}(\Omega_2) \neq \emptyset$, $\text{dom}(\Omega_1) \cap \text{dom}(\Omega_3) = \emptyset$ and $\text{dom}(\Omega_2) \cap \text{dom}(\Omega_3) = \emptyset$. Finally, recall that Ω_0 denotes the multiset consisting of exactly the empty mapping μ_0 .

We will show, for each axiom, examples of cases not satisfied by the negation operators. We will use a generic operator NEG to denote any of the operators used in our analysis, i.e. DIFF and MINUS.

Axiom (a): $(P_1 \text{ NEG } P_2) \equiv \emptyset$.

Both, DIFF and MINUS satisfy in all the cases.

Axiom (b): $(P_1 \text{ NEG } P_\emptyset) \equiv P_1$.

Both, DIFF and MINUS satisfy in all the cases.

Axiom (c): $(P_\emptyset \text{ NEG } P_1) \equiv \emptyset$.

Both, DIFF and MINUS satisfy in all the cases.

Axiom (d): $(P_1 \text{ NEG } (P_1 \text{ NEG } (P_1 \text{ NEG } P_2))) \equiv (P_1 \text{ NEG } P_2)$.

Both, DIFF and MINUS satisfy in all the cases.

Axiom (e): $((P_1 \text{ AND } P_2) \text{ NEG } P_2) \equiv P_\emptyset$.

- DIFF satisfies in all the cases.
- MINUS fails in four cases. For instance, if $\llbracket P_1 \rrbracket = \Omega_0$ and $\llbracket P_2 \rrbracket = \Omega_0$ then $\llbracket ((P_1 \text{ AND } P_2) \text{ MINUS } P_2) \rrbracket = \Omega_0$ instead of \emptyset .

Axiom (f): $((P_1 \text{ NEG } P_2) \text{ AND } P_2) \equiv P_\emptyset$.

- DIFF satisfies in all the cases.
- MINUS fails in eleven cases. For instance, if $\llbracket P_1 \rrbracket = \Omega_0$ and $\llbracket P_2 \rrbracket = \Omega_2$ then $\llbracket ((P_1 \text{ MINUS } P_2) \text{ AND } P_2) \rrbracket = \Omega_2$ instead of \emptyset .

Axiom (g): $(P_1 \text{ NEG } (P_1 \text{ AND } P_2)) \equiv (P_1 \text{ NEG } P_2)$.

- DIFF satisfies in all the cases.
- MINUS fails in seven cases. For instance, if $\llbracket P_1 \rrbracket = \Omega_1$ and $\llbracket P_2 \rrbracket = \Omega_0$ then $\llbracket (P_1 \text{ MINUS } (P_1 \text{ AND } P_2)) \rrbracket = \Omega_0$ whereas $\llbracket (P_1 \text{ MINUS } P_2) \rrbracket = \Omega_1$.

Axiom (h): $(P_1 \text{ AND } (P_1 \text{ NEG } P_2)) \equiv (P_1 \text{ NEG } P_2)$.

- Both, DIFF and MINUS fail under bag semantics, in five and twelve cases respectively.
- For instance, both operators fail when $\llbracket P_1 \rrbracket = \Omega_1$ and $\llbracket P_2 \rrbracket = \emptyset$ such that $\llbracket (P_1 \text{ AND } (P_1 \text{ NEG } P_2)) \rrbracket = \Omega_1 \bowtie \Omega_1$ whereas $\llbracket (P_1 \text{ NEG } P_2) \rrbracket = \Omega_1$.

Axiom (i): $((P_1 \text{ NEG } P_2) \text{ UNION } P_2) \equiv (P_1 \text{ UNION } P_2)$.

- DIFF generates distinct solutions in ten cases, and fails under bag semantics in four cases. For instance, if $\llbracket P_1 \rrbracket = \Omega_0$ and $\llbracket P_2 \rrbracket = \Omega_1$ then $\llbracket ((P_1 \text{ DIFF } P_2) \text{ UNION } P_2) \rrbracket = \Omega_1$ whereas $\llbracket (P_1 \text{ UNION } P_2) \rrbracket = \Omega_0 \cup \Omega_1$.
- MINUS fails under bag semantics in three cases. For instance, if $\llbracket P_1 \rrbracket = \Omega_1$ and $\llbracket P_2 \rrbracket = \Omega_1$ then $\llbracket (((P_1 \text{ MINUS } P_2) \text{ UNION } P_2)) \rrbracket = \Omega_1$ whereas $\llbracket (P_1 \text{ UNION } P_2) \rrbracket = \Omega_1 \cup \Omega_1$.

Axiom (j): $((P_1 \text{ UNION } P_2) \text{ NEG } P_2) \equiv (P_1 \text{ NEG } P_2)$.

- DIFF satisfies in all the cases.
- MINUS fails in four cases. For instance, if $\llbracket P_1 \rrbracket = \Omega_1$ and $\llbracket P_2 \rrbracket = \Omega_0$ then $\llbracket (((P_1 \text{ UNION } P_2) \text{ MINUS } P_2)) \rrbracket = \Omega_1 \cup \Omega_0$ whereas $\llbracket (P_1 \text{ MINUS } P_2) \rrbracket = \Omega_1$.

Axiom (k): $(P_1 \text{ NEG } (P_1 \text{ AND } P_3)) \equiv ((P_1 \text{ NEG } P_1) \text{ UNION } (P_1 \text{ NEG } P_3))$.

- DIFF fails in ten cases under bags semantics. For instance, if $\llbracket P_1 \rrbracket = \Omega_1$, $\llbracket P_2 \rrbracket = \Omega_0$ and $\llbracket P_3 \rrbracket = \Omega_2$ then $\llbracket ((P_1 \text{ DIFF } (P_1 \text{ AND } P_3))) \rrbracket = \Omega_2$ whereas $\llbracket (((P_1 \text{ DIFF } P_1) \text{ UNION } (P_1 \text{ DIFF } P_3))) \rrbracket = (\Omega_2 \setminus \Omega_1) \cup \Omega_2$.
- MINUS generates distinct solutions in twenty two cases, and fails under bag semantics in sixty five cases. For instance, if $\llbracket P_1 \rrbracket = \Omega_0$, $\llbracket P_2 \rrbracket = \Omega_1$ and $\llbracket P_3 \rrbracket = \Omega_1$ then $\llbracket ((P_1 \text{ MINUS } (P_1 \text{ AND } P_3))) \rrbracket = \Omega_0$ whereas $\llbracket (((P_1 \text{ MINUS } P_1) \text{ UNION } (P_1 \text{ MINUS } P_3))) \rrbracket = \Omega_1$.

Axiom (1): $(P_1 \text{ NEG}(P_1 \text{ UNION } P_3)) \equiv ((P_1 \text{ NEG } P_1) \text{ AND}(P_1 \text{ NEG } P_3))$

.

- Under bag semantics, DIFF and MINUS fail in seven and forty eight cases respectively.
- For instance, both operators fail when $\llbracket P_1 \rrbracket = \Omega_0$, $\llbracket P_2 \rrbracket = \Omega_0$ and $\llbracket P_3 \rrbracket = \Omega_1$ such that $\llbracket (P_1 \text{ NEG}(P_1 \text{ UNION } P_3)) \rrbracket = \Omega_1$ and $\llbracket ((P_1 \text{ NEG } P_1) \text{ AND}(P_1 \text{ NEG } P_3)) \rrbracket = \Omega_1 \bowtie \Omega_1$.